Unstructured Mesh Computations on CCMs [1]

Mark T. Jones [a] Karthik Ramachandran [b]

[a] *Virginia Tech*
[b] *Intel Corp*

Abstract

Configurable Computing Machines (CCMs) have been able to provide orders of magnitude increases in execution rates for applications such as image processing, signal processing, and automatic target recognition. This paper describes the use of CCMs to accelerate complex, large-scale scientific computations. These applications present a challenge for CCMs because of their large size, hundreds of thousands of lines of code, and the unstructured nature of the computations. This paper describes strategies for accelerating scientific computations on CCMs and demonstrates the effectiveness of one such strategy on the Annapolis Micro Systems WildForce board. Results from this implementation are analyzed.

Key words: configurable computing, floating point, finite element

1 Introduction

Configurable computing machines (CCMs) have proven very effective for accelerating applications such as signal processing, image processing, and automatic target recognition [2,6,34,40,41]. Commercial CCMs, which use FPGAs as computation elements, are well-suited for these well-structured applications in which bitwise operations dominate the computation. As FPGAs have become larger and faster, word-level operations such as floating point addition and multiplication have become practical [25,26,37,42]. Indeed, an early CCM, the Splash-2, was used for a low-precision implementation of a Poisson equation solver using finite differencing on a structured grid [28].

[1] This work is supported by National Science Foundation grants ASC-9501583 and CDA-9729893.

The contribution of this paper is the presentation of approaches for using CCMs to accelerate complex, large-scale scientific applications. Such applications rely on floating point computation in their computational kernels. In addition, these applications are typically very complex with hundreds of thousands of lines of code written by many different people. The approaches described in this paper focus on selecting and designing important computational kernels for these applications to implement on the CCM without requiring the re-coding of significant components of the application.

In the next two subsections the target CCMs are described along with the potential advantages and disadvantages of using them to accelerate computations. In Section 2, approaches for using a CCM to accelerate scientific computations on a CCM are discussed. The design and implementation of computational kernels on CCMs for supporting such scientific computations is examined in Section 3. Experimental results are presented and analyzed in Section 4. Finally, conclusions and plans for future work are given in Section 5.

1.1 Overview of Target CCMs

The approaches in this paper target current commercially available CCMs such as the WildForce board [39]. Other technologies such as the NAPA chip [35], the SLAAC board [30], and the Colt/Stallion chips [4] offer promising alternatives, but were not available for significant testing when this work was performed. Because of its widespread use, the focus in this paper is on the WildForce board as a model CCM. Like most available CCMs, this board uses field programmable gate array (FPGA) chips for the processing elements (PEs). Unless otherwise noted, references to CCMs in the remainder of the paper will refer to CCMs similar to the WildForce.

The WildForce board with five processing elements is shown in Figure 1. The board interfaces to a host PC via the PCI bus. The host CPU may communicate with the processing elements via either hardware FIFO queues on the board or by writing directly into the local memories of the PEs. The PEs may communicate with each other via a crossbar switch as well as nearest neighbor connections. Particularly important for the applications in this paper is the memory local to every PE; this local memory can take the form of either static or dynamic RAM. Each PE has a 32-bit data bus and 22-bit address bus dedicated to its local memory. This allows the PEs to concurrently access all five memories to provide significant memory bandwidth, a major benefit in many scientific computations.

In addition to the benefit of high memory-to-PE bandwidth, CCMs offer the benefit of custom construction of computational kernels. This allows for par-

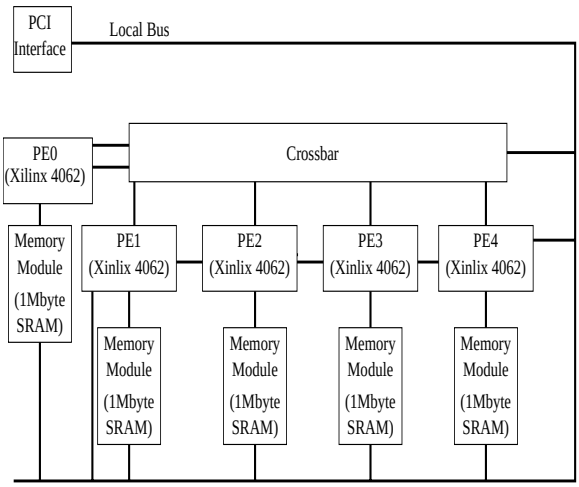

Fig. 1. A diagram of the WildForce CCM board illustrating the five processing elements, each with their own local memory.

allelism through the creation of multiple functional units on the PEs as well as custom pipelining to achieve high throughput. Further, unlike many new architectures, these CCMs don't require a new operating system with the expected initial instabilities. The CCM operates in the role of co-processor to the host CPU and is controlled by subroutines from a library produced by the vendor.

These CCMs do have drawbacks. The development effort for the construction of applications on a CCM is not as straightforward as writing a C program. [2] Currently, the development language of choice is the hardware description language VHDL [1]. CCMs are not closely coupled with the host CPU; they are typically connected to the PCI bus of the host machine. On most current PCs, the theoretical peak speed of the PCI bus is 132 megabytes per second [38], far short of the cache-to-CPU speeds on modern CPUs [19]. This lack of tight integration means that the communication between the host CPU and the CCM should be minimized. The NAPA chip [35] is a CCM research project that will provide tight coupling between a CPU and the configurable logic by putting them on the same chip; it is not clear, however, that this chip will have the same high aggregate memory bandwidth possible in an array of PEs that is so beneficial to the applications discussed in this paper. Finally, the memory local to a PE can only be accessed by that PE and the host CPU; there is no mechanism that allows one PE to directly access the memory of another PE. To implement a nonlocal memory reference capability would require the implementation of a functional unit on each PE that would accept memory access requests from other PEs and multiplex these requests with local PE requests for memory. While certainly feasible, this capability

[2] There are, however, several projects focusing on a high-level language compiler for CCMs [14,16,20,29,31].

would use valuable reconfigurable logic and make memory references more expensive. Thus, it is clearly advantageous to implement algorithms whose memory accesses are almost exclusively to memory local to a PE.

Current CCMs have much in common with SIMD parallel computers such as the Thinking Machines CM-2. For example, CCMs essentially consist of many small programmable units that can communicate with one another. Like SIMD computers, algorithms with a regular data access pattern typically perform well on CCMs; less structured data access patterns and programs are more difficult to map. CCMs have characteristics, however, that make algorithm design and implementation significantly different than on SIMD computers. The configurable logic in the FPGAs that comprise most CCMs is programmable at a lower level than the typical 1-bit processors in SIMD computers. This allows for more control, for example, over the construction of computational pipelines. Further, at this time, the programming tools and paradigms used are much different. Programs for CCMs are essentially described as logic components in much the same way that one constructs an application specific integrated circuit. The programmer is responsible for deciding which logic blocks are mapped to a particular PE and then a vendor-supplied utility is used to place and route these logic blocks on each of the PEs. In an SIMD machine, the programmer views the computer as an array of individual processors which must be programmed in parallel; typically, the same program is run on each individual processor with different data. Thus, the focus in SIMD programming is to describe a parallel program that can use the fixed computational element efficiently. The programming focus for CCMs is typically the construction of custom computational units that are, perhaps, replicated across the PEs. The programmer is concerned with constructing fast computational units that fit within the given amount of reconfigurable logic; the reconfigurable logic units are not typically viewed as individual processors that must be programmed.

1.2 Overview of Scientific Applications

The solution of partial differential equations (PDEs) is a required step in many scientific applications. The solution of systems of partial differential equations can be extremely computationally intensive, making it an excellent potential candidate for acceleration via CCMs. These systems of PDEs are usually constructed on a mesh that represents the physical domain. For geometrically simple domains, it is straightforward to create a *structured* mesh that conforms to that domain. In a structured mesh, the vertices are connected to their neighbors in a regular pattern across the entire mesh. Such structured meshes can be well-suited to implementation on a CCM because of this regular connection pattern; one simply performs the same computation on every vertex allowing for special cases only on the boundaries of the domain [28].

For more complex domains, however, the use of *unstructured* meshes can allow for meshes to be constructed for complex geometries as well as result in meshes with many fewer vertices than are present in the corresponding structured mesh. In an unstructured mesh, there is no regular connection pattern between the vertices. We give an example of both mesh types in Figure 2.

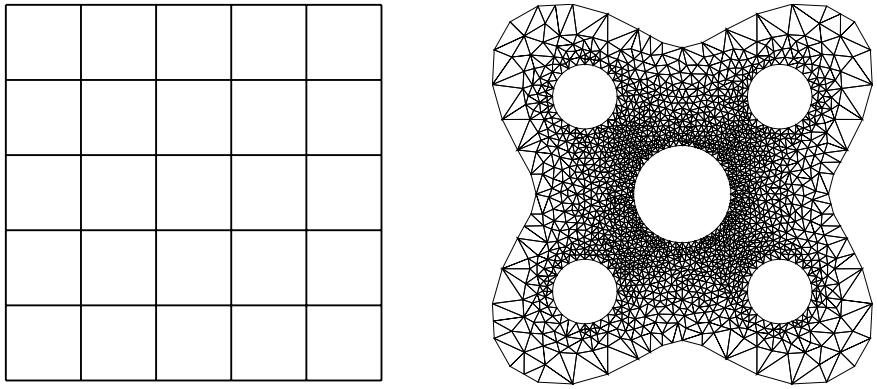

Fig. 2. On the left is a structured mesh with a regular local neighbor connection patter. On the right is an unstructured mesh that conforms to a complex geometry.

The finite element method (FEM) is method used to discretize PDEs [33] and is particularly well-suited for unstructured mesh computations. There are many software systems available for solving PDEs in various application domains using the FEM, both commercial and research-oriented. Such software packages typically require hundreds of thousands to millions of lines of code. These codes contain file i/o, user interfaces, and many other features that cannot practically be mapped onto FPGAs or related PEs; further, these features typically require the services of an operating system. Fortunately, the bulk of the execution time for these packages is typically spent in a small number of computational kernels that are typically not large and do not require operating system services. The approach in this paper will be to identify the kernels that are appropriate for CCM execution as well as determine the means for integrating the CCM-based kernels without incurring overwhelming penalties for data movement between the CCM and the host CPU.

2 Application Description and Integration

In this section a canonical finite element computation on an unstructured mesh is described. Computationally intensive components of the algorithms are identified. Finally, strategies for decomposing the algorithms across a host CPU and CCM are described.

A simplified high-level algorithm representative of finite element computa-

Read from a file(s) information describing the domain
Construct a mesh M_0 that conforms to the input geometry
Read from a file(s) problem specific information such as
    boundary conditions and material characteristics
$i = 0$
Repeat
    Assemble a sparse matrix, K_i, from the mesh, M_i
    Assemble a vector, f_i, from the mesh, M_i
    Solve the linear system $K_i u_i = f_i$
    Write to a file(s) information describing the current solution, u_i
    Estimate the error local to each element of M_i
    If maximum error estimate on M_i is too large **then**
        Based on the error estimates, refine M_i to get M_{i+1}
    Endif
    $i = i + 1$
Until maximum error estimate on M_i is satisfactory

Fig. 3. A simplified algorithm representative of a finite element computation on an unstructured mesh. Major operations are underlined.

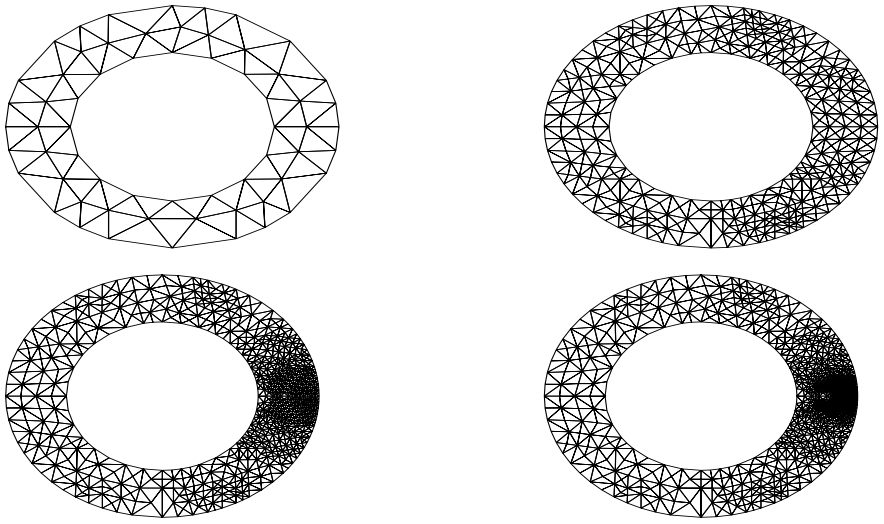

Fig. 4. A sequence of meshes generated by the procedure in Figure 3. The right side of the mesh is refined because the solution is rapidly changing in that region of the mesh.

tions on unstructured meshes is given in Figure 3; operations underlined in the figure are discussed in the following paragraphs. The three primary data structures associated with this algorithm are the mesh, M_i, the sparse matrix, K_i, and many vectors, including the unknowns, u_i, and the right-hand side, f_i. An unstructured mesh is often represented as a collection of elements (the triangles in Figure 4) and vertices. Virtually every operation in the finite element program will operate on this mesh. The sparse matrix is typically only acted upon during the assembly phase and the linear system solution phase,

yet these operations typically consume most of the compute cycles.

File operations, read and write, involving the input of problem parameters and the output of solution-related data fall clearly into the domain of general-purpose CPUs with operating system services. The CCM implementation of the remainder of the operations in Figure 3 is discussed in the following subsections. The proposed decomposition of operations and data structures assigns computationally intensive, localized operations to the CCM while keeping the complex data structures and data structure manipulation routines on the host CPU.

2.1 Mesh Manipulation on CCMs

The mesh manipulation operations, construct and refine, are more difficult to classify. The computation time for these operations can vary widely [13]; typically, the time is not insignificant, but is much smaller than the matrix assembly and linear system solution operations. Unstructured mesh construction algorithms [3,21] tend to be complex with memory access patterns that tend not to exhibit much locality and cannot be predicted *a priori*. For this reason, they are likely to be a poor choice for CCMs because (1) the available logic on most CCMs is inadequate to store complex programs efficiently,[3] (2) implementing such a complex program on a CCM would be a daunting task given currently available tools, and (3) the memory access patterns do not lend themselves to efficient implementation on CCMs. Mesh refinement algorithms, however, may have more potential, particularly if combined with error estimation in a CCM implementation. Typical mesh refinement algorithms [13] are operations that are local to an element and its immediate neighbors. A local error estimate is computed on an element, then the element is refined if the estimate is too high. Local error estimation techniques vary depending on the application, but most are simple routines that involve information local to the element and several floating point operations. Similarly, refining an element can be something as simple as bisecting a triangle. Complexities can arise if there is significant information stored with an element because this information must be processed when an element is refined; managing complex data structures is not a strength of current CCMs.

Given the complexity of constructing the mesh and the likelihood that complex data structures are used to store the mesh and related data, it appears that the storage of an entire unstructured mesh data structure in the local memory of the CCM is a poor design choice. Components of the mesh, however, could be written to the local memory of the CCM to facilitate operations such as

[3] Research projects in run-time reconfiguration provide a "virtual" program space on CCMs to allow the execution of large programs [18,36].

local error estimation. For example, many local error estimation functions need only the coordinates of the vertices and the estimated solution at each of the vertices to perform their task. The output of an error estimation function is typically a single floating point number.

2.2 Matrix/Vector Assembly on CCMs

Aspects of matrix/vector assembly have the potential for acceleration by CCMs. There are three major steps of matrix assembly: (1) the determination of the structure of the global sparse matrix based on the mesh, (2) the evaluation of the local element matrices, and (3) the assembly of the local element matrices into the global sparse matrix structure. An illustration of the relation between the sparse matrix, local element matrices, and the underlying mesh is given in Figure 5. As we will note later in this section, step (3), the assembly of the sparse matrix, may not be required for some linear system solution methods. Vector assembly is not nearly as complex. Because a vector is not sparse, the assembly of a vector only requires the evaluation of the local element vectors and the assembly of these local vectors into a global vector.

The determination of the global sparse matrix is a fairly complex operation that requires traversing the mesh, examining the neighborhoods of vertices, and allocating the correct data structure storage for the sparse matrix. The complexity of the operation mediates against the implementation on a CCM; further, the previous subsection illustrated the need to store the mesh data structure on the host CPU.

The evaluation of local element matrices and local element vectors is, however, a good candidate for CCM implementation. The element matrix evaluation routines are application specific, but typically involve a moderate number of floating point operations on small matrices with the most time-consuming portion being a matrix-by-matrix multiplication. Typical sizes for the resulting element matrices range from 3×3 to 20×20. The element vector evaluation routines typically require less computation because they operate on vectors of the same order as the matrices. These subroutines are typically simple enough for straightforward implementation on a CCM. The typical input requirements are the coordinates of the vertices and, perhaps, data representing the physical properties of the elements. [4] The output from an element matrix evaluation subroutine is the local element matrix.

The assembly of the local element matrices into the sparse matrix data structure is a moderately complex operation. Each entry in the local element matrix

[4] Note that, of course, some applications may require much more complex elements that require more input data and complex evaluation subroutines.

must be added to the appropriate entry of the global sparse matrix; the complexity arises because the appropriate location in the global sparse matrix must be found. This search requires that indirect addresses and integer comparisons must be performed. This is well within the range of complexity suitable for a CCM, but is more difficult than the purely local element-based operations. Depending on the type of linear system solution chosen, the assembly into the global sparse matrix may not be required. Assembly of the global vector from the local element vectors is a straightforward operation for implementation on CCMs because no sparse matrix data structure is involved.

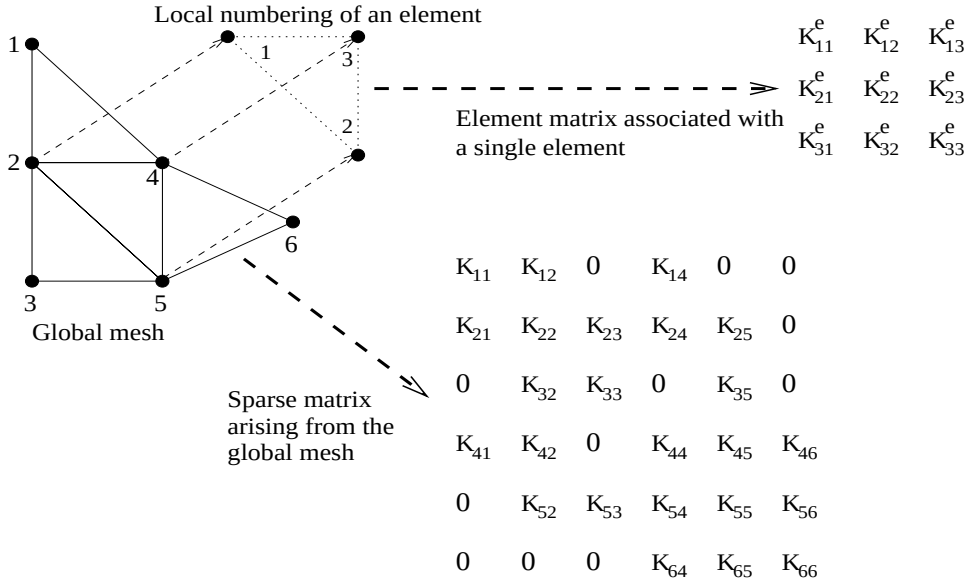

Fig. 5. The figure is a depiction of the local element matrix and the assembled sparse matrix for a mesh with 3 unknowns per element. Note that the local element matrices are 3×3. Each entry in the local element matrix is added to the corresponding entry in the assembled sparse matrix.

2.3 Sparse Linear System Solution on CCMs

There are many methods used to solve the sparse linear systems arising from the FEM. These methods include direct methods, which use matrix factorization [11,12,15], as well as iterative methods which repeatedly improve an estimate to the solution [17,22,27,43]. The focus of this paper will be on accelerating iterative solution methods for symmetric matrices, however, the same approaches can be used successfully for many direct solution methods as well as for nonsymmetric problems. The most successful iterative method for symmetric matrices is the conjugate gradient method [17], particularly, the preconditioned conjugate gradient method[7]. The conjugate gradient algorithm is given in Figure 6. The most computationally demanding aspect of the algorithm is the multiplication of the sparse matrix by a vector. This operation is performed every iteration; a typical matrix solution may require

hundreds of iterations.

$x_0 = 0$
$r_0 = b$
$k = 1$
While $r_{k-1} > tol$ **do**
  $\beta_k = (r_{k-1}, r_{k-1})/(r_{k-2}, r_{k-2})$ $\quad (\beta_1 \equiv 0)$
  $p_k = r_{k-1} + \beta_k p_{k-1}$ $\quad (p_1 \equiv 0)$
  $t_k = Ap_k$
  $\alpha_k = (r_{k-1}, r_{k-1})/(p_k, t_k)$
  $x_k = x_{k-1} + \alpha_k p_k$
  $r_k = r_{k-1} - \alpha_k t_k$
  $k = k + 1$
EndWhile
$x = x_n$

Fig. 6. The conjugate gradient algorithm solves the linear system of equations $Ax = b$ for x. Note (x, y) denotes the inner product of the vectors x and y.

The conjugate gradient algorithm or selected components of the algorithm are excellent candidates for implementation on a CCM. The input to the algorithm is the global sparse matrix and the output is a single vector. The operations in the algorithm are several scalar operations, vector-vector operations, such as inner products, and the sparse matrix by vector multiplication. All of the operations can be accomplished via multiplication and addition except for two scalar divisions and the comparison required by the **While** construct. In fact, if the host CPU controls the iterations, it could perform the comparison and the divisions while exchanging only a few scalar values with the CCM.

A second approach would be to perform only the sparse matrix by vector multiplication on the CCM and assign the remainder of the operations to the host CPU. This would require the sparse matrix to reside on the CCM; each iteration of conjugate gradient would require (1) the host CPU to write the vector, p_k, to the CCM, (2) the CCM to perform the matrix by vector multiplication to produce t_k, and (3) the host CPU to read the vector, t_k, from the CCM. This approach allows for a simpler implementation at the cost of more communication between the host CPU and the CCM.

2.4 The Proposed Task Decomposition

The proposed approach retains complex data structures and operations that modify those data structures on the host CPU. The unstructured mesh data structure as well as the relationship between the unstructured mesh and the sparse matrix are kept in the memory of the host CPU. The CCM is responsible for compute intensive operations on individual elements; for example,

the evaluation of the local element matrices. The element matrices are stored in the memory of the CCM and never need to be transferred to the host CPU. The overall approach is given in graphical form in Figure 7. Note that the communication required between the host CPU and the CCM is relatively small compared to the computation that takes place on the CCM. This computation-to-communication ratio will be discussed in more detail in the next section.

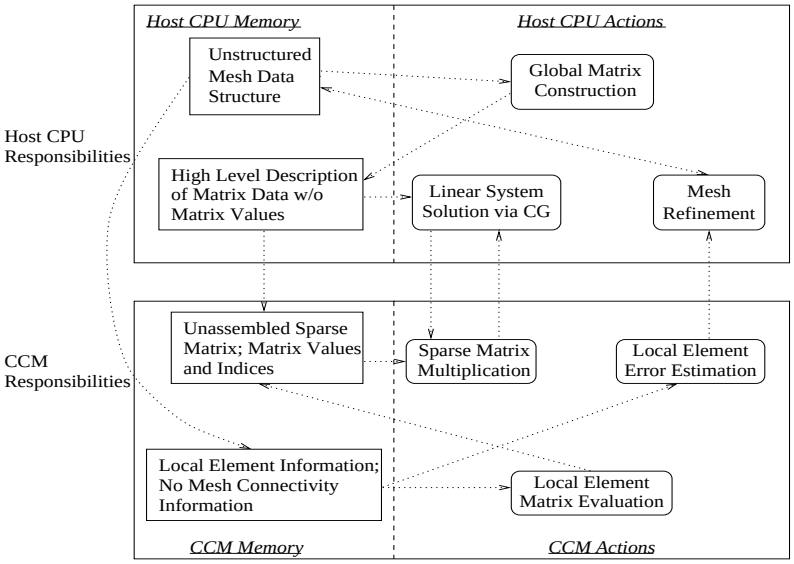

Fig. 7. The top of the figure gives the data and computations that remain on the host CPU; the bottom shows the data and computations on the CCM. The communication required between the host CPU and the CCM is shown in the dotted lines.

Note that this type of computation, organized in phases, is ideal for taking advantage of the reconfigurable nature of the PEs. The PEs can be reconfigured to use all of the available resources for each phase of the computation. Because each of the phases takes a significant amount of time to execute, the time for reconfiguration is amortized. There is also a potential for partial reconfiguration, because each phase requires, for example, a multiply-accumulator (MAC) unit.

3 Computational Kernel Design

In this section, a description is given of the implementation of the sparse matrix-by-vector multiplication mentioned in Section 3 on a target CCM, the Annapolis Micro Systems WildForce board [39]. The sparse matrix-by-vector multiplication implementation was selected as the focus of the implementation efforts because it is the most time-consuming component of the application

and highlights the use of the MAC unit that forms the core of most of the required CCM operations.

This section describes sparse matrix-by-vector multiplication, $Kx = y$, where K is an unassembled matrix, x is the input vector, and y is the output vector. The unassembled sparse matrix can be stored as an array of small dense matrices along with the global indices, *index*, for each of those matrices. Each small dense matrix, K^e, can be considered as a separate operation; for example, consider the operation for a $3x3$ matrix associated with the highlighted element in Figure 5. The global indices of the highlighted element are 2, 4, and 5 corresponding to the local indices 1, 2, and 3. The contribution of that element to the result of the matrix-by-vector multiplication is

$$\begin{aligned} y_2 &= K^e_{11}x_2 + K^e_{12}x_4 + K^e_{13}x_5 + y_2 \\ y_4 &= K^e_{21}x_2 + K^e_{22}x_4 + K^e_{23}x_5 + y_4 \\ y_5 &= K^e_{31}x_2 + K^e_{32}x_4 + K^e_{33}x_5 + y_5, \end{aligned} \tag{1}$$

where this operation is denoted in general as

$$y[index_e] = K^e x[index_e] + y[index_e]. \tag{2}$$

The overall unassembled sparse matrix operation can be expressed as in Figure 8.

$y = 0$
For $k = 1$ to the number of elements **do**
    $y[index_i] = K^i x[index_i] + y[index_i]$
EndFor

Fig. 8. The general procedure for the multiplication of an unassembled matrix by a vector.

The CCM algorithm for this unassembled sparse matrix operation allocates an equal number of the element matrices to each of the five PEs on the WildForce board. This partitioning implies that each PE will contribute a *part* of y and that each PE will need only those x_i that are required for the matrices assigned to it. The algorithm for five PEs is given in Figure 9.

The algorithm in Figure 8 is executed independently by each of the PEs for the set of element matrices assigned to it. The core of the PE algorithm is a matrix-by-vector multiplication unit that multiplies the small dense element matrix by the appropriate entries in the vector x, and adds them to its copy of vector y as in Equation 1. The reconfigurable nature of the PEs is exploited by constructing matrix-by-vector multiplication units that are optimized for specific matrix sizes. The correct unit can be downloaded depending on the particular elements in a mesh. In a complex mesh with differing element sizes,

Assign a set of matrices to each of the five PEs
For $i = 1 \ldots 5$ **do**
    Host CPU writes required elements of x into PE i via DMA
    Host CPU signals PE i to begin multiplication
EndFor
Host CPU initializes its copy of the y vector to 0
For $i = 1 \ldots 5$ **do**
    Host CPU waits for completion signal from PE i
    Host CPU reads (via DMA) PE i's contribution to the y vector
    Host CPU adds this contribution to its copy of the y vector
EndFor

Fig. 9. The overall algorithm for sparse matrix-by-vector multiplication by five CCM PEs under the control of the host CPU.

different PEs could be responsible for different element sizes; one would expect some loss in efficiency in that case because it is unlikely that the proportions of element sizes would match a multiple of the number of PEs.

The dense matrix-by-vector multiplication unit has a pipelined MAC unit at its core. This MAC unit has a 3-stage floating point multiplier and a 3-stage floating point adder. The storage format of the 32-bit floating point numbers is as described in IEEE 754 [5], but the unit does not handle exceptions and rounding as specified in the standard. For a particular $m \times m$ dense element matrix, these units must carry out the computation as described in Figure 10. In the implementation described here, the operations in Figure 10 are pipelined such that there is overlap between the operations, but there is no overlap between matrices.

1. Read the m indices, $index_y$, for the y vector
2. Read the m initial values of y indicated by $index_y$
3. Read the m indices, $index_x$, for the x vector
4. Read the m values of x indicated by $index_x$
5. Read the m^2 entries of the matrix A
6. Carry out m^2 multiplications and m^2 additions
7. Write the m new values of y to the locations indicated by $index_y$

Fig. 10. The PE algorithm for multiplying a dense element matrix by the appropriate values in x and producing the appropriate values of y. Note that there is significant overlap in the execution of the steps due to the use of pipelining.

The characteristics of the dense matrix-by-vector multiplication units are given in Table 1 for different matrix sizes. Note that the throughput of the units increases as the matrix size increases. This increase is explained by the expression for the throughput (in floating point operations per clock cycle)

$$\frac{2m^2}{m^2 + 5m + C}, \tag{3}$$

where the term $m^2 + 5m$ is the time required to read/write the matrix/vector elements from memory local to the PE and C is a constant associated with the length of the pipeline as well as read/write delays. The FPGA resources required also increase as the size of the matrix increases because of the use of temporary registers; these temporary registers serve as a cache to reduce the i/o requirements of the unit. These units run at 15MHz on the Xilinx 4062XL chip. The bottleneck is the MAC unit; methods for improving the throughput of this unit are described in [32].

Table 1
The resource requirements (as a percentage of the total available) and throughput for the dense matrix-by-vector multiplication units are given below as implemented on a Xilinx 4062XL FPGA running at 15MHz.

Matrix Size	CLBs	Latches	Flip/flops	4-LUTs	3-LUTs	Mflops
3×3	61%	1%	15%	52%	10%	6.92
4×4	61%	1%	17%	51%	13%	9.60
6×6	73%	1%	20%	60%	17%	13.85

4 Experimental Results

Results from the use of the sparse matrix by vector multiplication used in an unstructured finite element computation are given in this section. These results are compared to a highly efficient general purpose CPU implementation of the same algorithm. Finally, the results are analyzed and a performance bottleneck is identified.

The application considered in this section is the solution of Poisson's equation using the finite element method in two and three dimensions. A Gaussian source term centered at a point is applied; this requires the use of adaptive refinement to effectively resolve the solution around the source as shown in Figure 4. [5] Three variants of these problem are considered. *Problem One* is a finite element model in two dimensions using linear basis functions; the element matrices in this problem are 3×3. *Problem Two* is identical to *Problem One* except that quadratic basis functions are used resulting in 6×6 element matrices. *Problem Three* is a finite element model in three dimensions using linear basis functions; the resulting element matrices are 4×4.

The execution rates for the entire sparse matrix by vector multiplication routine in Figure 9, including CPU host overhead, are given in Table 2. The rates for different numbers of PEs are compared to the time to execute the same

[5] More details on this problem can be found in [23].

computation on a 300 MHz Pentium II processor using the optimized BLAS [9,10,24] operations in the Intel Performance Library Suite [8]. In each of the three problems, the WildForce board is faster than the Pentium II.

Table 2
A comparison of the sparse matrix by vector multiplication running on the WildForce board against the same algorithm on a 300 MHz Pentium II. The number of element matrices in each case is given; the number of element matrices per PE is kept roughly constant. The number of words transferred per sparse matrix by vector multiplication when the host CPU reads or writes vectors is given in the last column. Comparisons are given for each of the three problems.

Problem	Platform	Number of matrices	Throughput (Mflops)	Words transferred
One	CCM (1 PE)	13,372	5.77	20,388
One	CCM (3 PEs)	39,879	14.9	76,419
One	CCM (5 PEs)	66,240	22.4	123,974
One	Pentium	66,240	9.70	N/A
Two	CCM (1 PE)	4,504	10.8	27,459
Two	CCM (3 PEs)	13,433	25.7	95,421
Two	CCM (5 PEs)	22,730	35.7	161,859
Two	Pentium	22,730	24.4	N/A
Three	CCM (1 PE)	8,751	8.37	5,568
Three	CCM (3 PEs)	28,659	22.8	36,570
Three	CCM (5 PEs)	50,475	33.8	88,107
Three	Pentium	50,475	16.8	N/A

Although this algorithm executes faster on the WildForce board than on the 300 MHz Pentium II, there is significant room for performance enhancement. As mentioned in Section 3, the throughput of the MAC units can be increased. More significant, however, is the overhead due to processing on the host CPU in the algorithm in Figure 9. This overhead arises because of the need for the CPU and CCM to exchange the x and y vectors. The number of words exchanged can be significant as shown in Table 2, however, the DMA transfer rates are high enough such that the time to actually transfer the vectors is not significant. However, the host CPU must gather the x vector before writing it to the CCM and the host CPU must add each PE's contribution to the y vector. Because there is only one CPU, these actions are serialized and, as the number of PEs used increases, they become a significant drag on performance. The breakdown of the runtimes in terms of CCM computation, DMA time, and CPU overhead is given in Table 3.

Table 3
The percentage of execution time is given for performing matrix multiplication, transferring vectors via DMA, and host CPU overhead.

Problem	Number of PEs	Proportion spent in multiplication	Proportion spent in DMA transfer	Proportion spent CPU overhead
One	1	79.09	1.04	19.87
One	3	56.64	2.80	40.56
One	5	46.03	3.69	50.28
Two	1	65.57	1.67	32.66
Two	3	38.90	3.94	57.16
Two	5	32.21	4.81	62.98
Three	1	58.98	0.34	40.68
Three	3	50.44	1.70	47.86
Three	5	42.32	3.29	54.39

Clearly the CPU overhead is the significant bottleneck preventing higher performance. If this bottleneck is removed, then a faster MAC unit or multiple MAC units can be used effectively. Section 3 describes an approach that will increase performance; if the bulk of the CG algorithm is implemented on the WildForce board, then the need for the CPU to manage the x and y vectors is alleviated. This will require a more complex implementation, but the resource consumption values in Table 1 indicate there is room on each PE for a more complex implementation. The two division operations in the CG algorithm would likely need to be implemented by the CPU to save on space; the overhead associated with that exchange of information would be minimal.

5 Conclusions and Future Work

CCMs have the potential to significantly accelerate unstructured mesh applications without requiring an overly complex implementation. Much of the complexity of these applications can be managed on the host CPU while still allowing the CCM to perform the bulk of the computation. The reconfigurable nature of the CCM can be used to advantage by reconfiguring the CCM for different phases of the computation as well as by providing custom modules for specific classes of unstructured meshes.

Experimental results show that CCMs are capable of accelerating a major computational kernel, sparse matrix-by-vector multiplication, for such applications. The CCM implementation was shown to be significantly faster than

the same algorithm on a 300 MHz Pentium II. The CCM implementation has significant room for improved performance; means for improving this algorithm were suggested in Section 4.

The potential for the use of CCMs in unstructured mesh applications has been shown via a proof-of-concept demonstration in this paper, but clearly more work remains to realize the full potential. New commercial FPGA technology provides sixteen times more reconfigurable logic and substantially faster clock speeds than were available on the WildForce platform for this implementation. This would lead to much faster multiply-accumulate units as well as many more multiply-accumulate units per chip. The implementation of the MACs can also be improved through deep pipelining rather than the simple three-stage pipeline used here. To extend the proof-of-concept implementation, the preconditioned conjugate gradient algorithm should be implemented with a substantial portion of the algorithm on the CCM. This would boost performance rates as well as reduce overall solution time by decreasing the number of iterations required in the CG method. A relatively clear path exists for substantially increasing performance while providing something beyond a proof-of-concept implementation.

References

[1] J. Armstrong and F. Gray, *Structured Logic Design with VHDL*, Prentice Hall, Upper Saddle River, NJ, 1993.

[2] P. Athanas and L. Abbott, *Processing images in real time on a custom computing platform*, IEEE Computer, 28 (1995), pp. 16–24.

[3] W. Barry, M. Jones, and P. Plassmann, *Parallel adaptive mesh refinement techniques for plasticity problems*, Advances in Engineering Software, to appear (1998).

[4] R. Bittner and P. Athanas, *Wormhole run-time reconfiguration*, in ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, Feb. 1997, pp. 79–85.

[5] I. Board, *IEEE standard for binary floating-point arithmetic*, ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.

[6] H. Chow, H. Alnuweiri, and S. Casselman, *FPGA-based transformable computers for fast digital signal processing*, in The Fourth IEEE Symposium on FPGAs for Custom Computing Machines, Napa, CA, April 1995, pp. 197–203.

[7] P. Concus, G. Golub, and D. O'Leary, *A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations*, in Sparse Matrix Computations, Academic Press, New York, 1976.

[8] I. Corporation, *Intel Math Kernal Library Reference Manual*, Portland, OR.

[9] J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software, 16 (1990), pp. 1–17.

[10] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, *An extended set of Fortran basic linear algebra subprograms*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1–32.

[11] I. Duff and J. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.

[12] I. S. Duff, *Direct methods for solving sparse systems of linear equations*, SIAM Journal of Scientific and Statistical Computing, 5 (1984), pp. 605–619.

[13] L. A. Freitag, M. T. Jones, and P. E. Plassmann, *The scalability of mesh improvement algorithms*, in IMA Volumes in Mathematics and its Applications, vol. 105, 1998, pp. 185–212.

[14] D. Galloway, *The transmorgrifier C hardware description language and compiler for FPGAs*, in IEEE Workshop on RPGAs for Custom Computing Machines, Napa, CA, April 1993, pp. 94–101.

[15] J. A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[16] M. Gokhale and R. Minnich, *FPGA computing in a data parallel C*, in IEEE Workshop on RPGAs for Custom Computing Machines, Napa, CA, April 1993, pp. 94–101.

[17] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952), pp. 409–436.

[18] R. Hudson, D. Lehn, and P. Athanas, *A run-time reconfigurable engine for image interpolation*, in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, J. Arnold and K. L. Pocek, eds., Napa, CA, April 1998.

[19] Intel Corporation, *Technical Product Brief for the Pentium II Overdrive Processor*, Santa Clara, CA, 1998.

[20] C. Iseli and E. Sanchez, *A C++ compiler for FPGA custom execution units synthesis*, in The Fourth IEEE Symposium on FPGAs for Custom Computing Machines, Napa, CA, April 1995, pp. 173–179.

[21] M. Jones and P. Plassmann, *Adaptive refinement of unstructured finite-element meshes*, Journal of Finite Elements in Analysis and Design, 25 (1997), pp. 41–60.

[22] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear systems*, Parallel Computing, 20 (1994), pp. 753–773.

[23] ——, *Parallel algorithms for adaptive mesh refinement*, SIAM Journal on Scientific Computing, 18 (1997), pp. 686–708.

[24] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Transactions on Mathematical Software, 5 (1979), pp. 308–325.

[25] Y. Li and W. Chu, *Implementation of single precision floating point square root on FPGAs*, in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April 1997, pp. 271–277.

[26] W. B. Ligon, S. McMillan, G. Monn, F. Stivers, and K. D. Underwood, *A re-evaluation of the practicality of floating-point operations on FPGAs*, in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, J. Arnold and K. L. Pocek, eds., Napa, CA, April 1998.

[27] T. A. Manteuffel, *An incomplete factorization technique for positive definite linear systems*, Mathematics of Computation, 34 (1980), pp. 473–497.

[28] K. Paar, *A custom computing machine solution for simulation of discretized domain physical systems*, master's thesis, Department of Electrical Engineering, Virginia Polytechnic Institute and State University, 1996.

[29] I. Page, *Constructing hardware-software systems from a single description*, 1996.

[30] R. Parker and B. Schott, *SLAAC home page*. http://www.east.isi.edu/SLAAC/contents.htm, 1998.

[31] J. B. Peterson, *A retargetable ANSI-C compiler for conifgurable computing machines*. http://www.ee.vt.edu/ ccm/, 1998.

[32] K. Ramachandran, *Unstructured finite element computations on reconfigurable computers*, master's thesis, Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, 1998.

[33] J. Reddy, *An Introduction to the Finite Element Method*, McGraw-Hill, New York, 1993.

[34] M. Rencher and B. L. Hutchings, *Automated target recognition on Splash-II*, in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April 1997, pp. 271–277.

[35] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, *The NAPA adaptive processing architecture*, in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, J. Arnold and K. L. Pocek, eds., Napa, CA, April 1998.

[36] N. Shirazi, W. Luk, and P. Y. Cheung, *Automating production of run-time reconfigurable designs*, in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, J. Arnold and K. L. Pocek, eds., Napa, CA, April 1998.

[37] N. Shirazi, A. Walters, and P. Athanas, *Quantitative analysis of floating point arithmetic on FPGA based cutom computing machines*, in The Fourth IEEE Symposium on FPGAs for Custom Computing Machines, Napa, CA, April 1995, pp. 155–162.

[38] W. Stallings, *Computer Organization and Architecture*, Prentice Hall, Upper Saddle River, NJ, 1996.

[39] A. M. Systems, *Wildforce reference manual*, 1998.

[40] J. Villasenor, B. Schoner, K. Chia, and C. Zapata, *Configurable computing solutions for automatic target recognition*, in Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, J. Arnold and K. L. Pocek, eds., Napa, CA, April 1996, pp. 70–79.

[41] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, *Programmable active memories: Reconfigurable systems come of age*, IEEE Transactions on VLSI, 4 (1996), pp. 56–69.

[42] A. L. Walters, *A scalable FIR filter implementation using 32-bit floating-point complex arithmetic on a FPGA based custom computing platform*, master's thesis, Department of Electrical Engineering, Virginia Polytechnic Institute and State University, 1997.

[43] R. White, *Multi-splitting of a symmetric positive definite matrix*, SIAM Journal of Matrix Analysis and Applications, 11 (1990), pp. 69–82.